UNIT - II

1Q. Explain JSON handling in Node.js with examples of parsing and stringification.

Answer:

JSON stands for **JavaScript Object Notation**. It is a **lightweight, text-based data interchange format** used to store and exchange data between a client and a server. JSON is easy for humans to read and write and easy for machines to parse and generate. It is language-independent, but it is derived from JavaScript syntax. Because of its simplicity and efficiency, JSON has become the most widely used data format in modern web applications.

In server-side programming using **Node.js**, JSON plays a very important role. Node.js applications frequently exchange data with web browsers, mobile applications, databases, and external APIs, and JSON is the standard format used for this communication.

"JSON represents data in the form of **key–value pairs**. Keys are strings, and values can be strings, numbers, objects, arrays, booleans, or null."

Example of JSON Data:

```
{
  "id": 101,
  "name": "Ravi",
  "branch": "CSE",
  "marks": 85,
  "isPassed": true
}
```

Characteristics of JSON

- JSON data is stored in **text format**
- It is **lighter than XML**
- It supports **nested objects and arrays**
- It is **platform and language independent**
- It is the default data format for **REST APIs**

Role of JSON in Node.js

Node.js is mainly used for building **server-side and network applications**. In Node.js:

- Client requests are received in JSON format
- Server responses are sent in JSON format
- Data is stored or transferred using JSON
- APIs use JSON as input and output

Node.js provides **built-in support** for JSON handling through the global `JSON` object.

JSON Object in Node.js

Node.js uses a global object called `JSON` which provides two important methods:

1. `JSON.parse()`
2. `JSON.stringify()`

These two methods are the backbone of JSON handling in Node.js applications.

JSON Parsing in Node.js

Definition of JSON Parsing

JSON parsing is the process of converting a **JSON string** into a **JavaScript object** so that it can be accessed and manipulated programmatically in Node.js.
In real-time applications, data received from:

- HTTP requests
- API responses
- Files
- Databases

is usually in **string format**, and it must be parsed before use.

Syntax of JSON.parse():

JSON.parse(jsonString)

- `jsonString` → A valid JSON formatted string
- Returns → JavaScript object

Example: JSON Parsing in Node.js :

```
const jsonData = '{"id":1,"name":"Anil","course":"Node.js"}';
const obj = JSON.parse(jsonData);

console.log(obj.id);
console.log(obj.name);
console.log(obj.course);
```

Output:
1
Anil
Node.js

Parsing JSON from a File in Node.js :

```
const fs = require("fs");
const data = fs.readFileSync("student.json", "utf8");
const student = JSON.parse(data);
consolelog(student.name);
```

Here, JSON data is read from a file and parsed into a JavaScript object for further processing.

JSON Stringification in Node.js

Definition of JSON Stringification

JSON stringification is the process of converting a **JavaScript object** into a **JSON string** so that it can be:

- Sent as a response to the client
- Stored in a file
- Transmitted over a network
- Stored in a database

Syntax of JSON.stringify():

JSON.stringify(object)

- `object` → JavaScript object
- Returns → JSON formatted string

Example: JSON Stringification

```
const student = {
  id: 102,
  name: "XYZ",
  department: "IT",
  marks: 90
};
const jsonString = JSON.stringify(student);
console.log(jsonString);
```

Output :

```
{"id":102,"name":"XYZ","department":"IT","marks":90}
```

Here, a JavaScript object is converted into a JSON string using JSON.stringify(). This string can be sent to a client browser or stored in a file.

Writing JSON Data to a File

```
const fs = require("fs");
const data = {
  productId: 501,
  productName: "Laptop",
  price: 55000
};
fs.writeFileSync("product.json", JSON.stringify(data));
```

This example shows how Node.js stores structured data into a JSON file.

JSON in HTTP Response (Node.js Server)

Example:

```
const http = require("http");
const server = http.createServer((req, res) => {
  const user = { id: 1, name: "Admin" };
  res.writeHead(200, { "Content-Type": "application/json" });
  res.end(JSON.stringify(user));
});
server.listen(3000);
```

In this example, Node.js sends a JSON response to the client browser.

JSON handling is a **core concept in Node.js development**. Using `JSON.parse()`, Node.js converts incoming JSON data into usable JavaScript objects, while `JSON.stringify()` converts JavaScript objects into JSON strings for transmission or storage. These operations are fundamental in building scalable, efficient, and data-driven server-side applications.

2Q: Explain the Buffer and Stream modules in Node.js. Compare streams vs buffers.

Answer:

In server-side JavaScript using **Node.js**, applications often deal with **large amounts of data** such as files, videos, audio streams, network packets, and database results. Handling such data efficiently is very important for performance and memory management. For this purpose, Node.js provides two powerful core concepts called **Buffer** and **Stream**.

Buffers and Streams are used to handle **binary data** and **data flow**, especially when working with file systems, network communication, and real-time applications. Understanding these modules is essential for building scalable and efficient Node.js applications.

Buffer Module in Node.js

Definition of Buffer

A **Buffer** in Node.js is a **temporary memory area** used to store **raw binary data**. It is mainly used when dealing with data that cannot be handled directly by JavaScript strings, such as images, audio files, video files, and network data.
Buffers are part of Node.js core and are designed to handle **binary streams of data outside the V8 heap memory**.

JavaScript was originally designed for browsers and works mainly with **Unicode strings**, not binary data. Since Node.js works at a lower level with files and networks, the Buffer module is required to:

- Handle binary data
- Improve performance
- Store raw data temporarily
- Interact with file systems and sockets

Creating a Buffer :

```
const buffer = Buffer.from("Hello Node.js");
console.log(buffer);
console.log(buffer.toString());
```

OutPut:

<Buffer 48 65 6c 6c 6f 20 4e 6f 64 65 2e 6a 73>

Hello Node.js

Here, the string is converted into a Buffer. Internally, the buffer stores data in **hexadecimal format**, and toString() converts it back into readable text.

Buffer from File Data:

```
const fs = require("fs");
const data = fs.readFileSync("data.txt");
console.log(data);          // Buffer
console.log(data.toString()); // Converted to string
```

Stream Module in Node.js

A **Stream** in Node.js is an **abstract interface** used to handle **continuous flow of data**. Instead of loading the entire data into memory at once, streams process data **chunk by chunk**, making them highly memory efficient.

Streams are widely used for handling:

- Large files
- Video/audio streaming
- Network communication
- Real-time data processing

Characteristics of Streams

- Data processed in **chunks**
- Memory efficient
- Suitable for **large data**
- Supports real-time processing
- Non-blocking and asynchronous

Types of Streams in Node.js

Node.js supports four types of streams:

1. **Readable Stream** – used to read data
2. **Writable Stream** – used to write data
3. **Duplex Stream** – both read and write
4. **Transform Stream** – modify data while passing

Example: Readable Stream

```
const fs = require("fs");
const readStream = fs.createReadStream("data.txt");
readStream.on("data", chunk => {
  console.log(chunk.toString());
});
```

In this program, the file is read in small chunks **instead of reading the full file at once. Each chunk is processed as soon as it is available.**

Example: Writable Stream:

```
const fs = require("fs");
const writeStream = fs.createWriteStream("output.txt");
writeStream.write("Welcome to Node.js Streams");
writeStream.end();
```

This example writes data gradually into a file using a writable stream.

Example using Piping Streams:

```
const fs = require("fs");
const readStream = fs.createReadStream("input.txt");
const writeStream = fs.createWriteStream("output.txt");
readStream.pipe(writeStream);
```

Piping allows data to flow automatically from one stream to another, improving performance and readability.

Real-Time Examples

Buffer Use Cases

- Reading small configuration files
- Handling binary packets
- Image processing
- Temporary data storage

Stream Use Cases

- Video streaming platforms
- File upload/download systems
- Live chat applications
- Large database exports

In Node.js, **Buffer** and **Stream** are essential modules for handling data efficiently. Buffers are used to store binary data temporarily in memory, while streams process data piece by piece without loading everything at once. For small data, buffers are sufficient, but for large and real-time data, streams are the best choice. Understanding the difference between buffers and streams helps developers build high-performance, scalable Node.js applications suitable for modern web systems.

3Q. Describe File System Operations in Node.js, including reading, writing, updating and deleting files.

Answer:

In server-side JavaScript using **Node.js**, applications often need to interact with files stored on the server. These operations include reading data from files, writing new data, updating existing content, and deleting files when they are no longer required. Such tasks are known as **file system operations**.

Node.js provides a built-in core module called the **File System (fs) module**, which allows applications to work with the operating system's file system efficiently. File system operations are widely used in real-time applications such as logging systems, configuration handling, data storage, and file upload/download services.

File System (fs) Module in Node.js

Definition

The **fs module** in Node.js provides an API to interact with files and directories on the server. It supports both **synchronous (blocking)** and **asynchronous (non-blocking)** operations, making it suitable for scalable and high-performance applications.

Importing fs Module

```
const fs = require("fs");
```

Types of File System Operations
Node.js supports the following major file system operations:

1. Reading files
2. Writing files
3. Updating files
4. Deleting files

Each operation can be performed using **synchronous** or **asynchronous** methods.

Reading Files in Node.js

File Reading:

Reading a file means retrieving the content stored in a file so that it can be processed or displayed by the application. File reading is commonly used for reading configuration files, JSON data, logs, and text files.

Asynchronous File Reading

```
const fs = require("fs");
fs.readFile("data.txt", "utf8", (err, data) => {
  if (err) {
    console.log("Error reading file");
  } else {
    console.log(data);
  }
});
```

In this method, Node.js reads the file **without blocking the execution** of other code. The callback function is executed once the file reading is completed. This approach is preferred in real-time applications.

Synchronous File Reading

```
const fs = require("fs");

const data = fs.readFileSync("data.txt", "utf8");

console.log(data);
```

This method blocks program execution until the file is completely read. It is simple but not recommended for large-scale or server-side applications.

Writing Files in Node.js

File Writing

Writing a file means creating a new file or inserting data into an existing file. Node.js allows writing data such as text, JSON, or binary content into files.

Asynchronous File Writing

```
const fs = require("fs");
fs.writeFile("sample.txt", "Welcome to Node.js", err => {
  if (err) {
    console.log("Error writing file");
  } else {
    console.log("File written successfully");
  }
});
```

If the file does not exist, it is created automatically. If it exists, the old content is overwritten.

Synchronous File Writing :

```
const fs = require("fs");

fs.writeFileSync("sample.txt", "Node.js File System");
```

Updating Files in Node.js

File Updating

Updating a file means modifying existing content by **adding new data** without deleting the old data. This is commonly required for logs, reports, and incremental data storage.

Appending Data to a File (Asynchronous):

```
const fs = require("fs");
fs.appendFile("sample.txt", "\nNew content added", err => {
  if (err) {
    console.log("Error updating file");
  } else {
    console.log("File updated successfully");
  }
});
```

The `appendFile()` method adds new content at the end of the file without removing existing data.

Appending Data Synchronously

```
fs.appendFileSync("sample.txt", "\nAdditional data");
```

Deleting Files in Node.js

File Deletion

Deleting a file means permanently removing it from the file system. This is required for cleanup operations, temporary files, and old data removal.

Asynchronous File Deletion

```
const fs = require("fs");
fs.unlink("sample.txt", err => {
  if (err) {
    console.log("Error deleting file");
  } else {
    console.log("File deleted successfully");
            }
});
```

Synchronous File Deletion

```
fs.unlinkSync("sample.txt");
```

File system operations form a fundamental part of Node.js applications. Using the **fs module**, Node.js provides powerful methods to read, write, update, and delete files efficiently. Asynchronous methods are preferred in server-side applications to ensure non-blocking execution and better performance. Proper understanding of file system operations helps in building reliable, scalable, and real-world Node.js applications.

4Q. Explain the steps in creating HTTP Web Server in Node.js using http module with example code.

Answer:

A **web server** is a software application that listens for client requests (usually from a web browser) and sends appropriate responses over the **HTTP protocol**. In modern web development, server-side applications must be **fast, scalable, and efficient**.

Node.js provides a built-in core module called the **http module**, which allows developers to create an HTTP web server without using any external libraries. Using this module, Node.js can handle thousands of concurrent client requests efficiently due to its **event-driven and non-blocking architecture**.

HTTP Module in Node.js

Definition

The **http module** in Node.js is a core module that enables Node.js applications to:

- Create HTTP servers
- Handle client requests
- Send HTTP responses
- Implement RESTful web services

Since it is a core module, it does not require any installation.

Basic Components of an HTTP Web Server

An HTTP web server in Node.js mainly consists of the following components:

1. HTTP module
2. Server creation
3. Request handling
4. Response generation
5. Listening on a port

HTTP Web Server in Node.js

```
const http = require("http");

const server = http.createServer((req, res) => {

  res.writeHead(200, { "Content-Type": "text/html" });

  res.write("<h1>Welcome to Node.js HTTP Server</h1>");

  res.end();

});

server.listen(3000, () => {

  console.log("Server is running at http://localhost:3000");

});
```

Steps to Create an HTTP Web Server in Node.js

Step 1: Import the HTTP Module

To create a web server, the first step is to import the built-in `http` module.

```
const http = require("http");
```

This statement loads the HTTP functionality into the application.

Step 2: Create the Server Using createServer()

The next step is to create a server using the `createServer()` method.

```
const server = http.createServer((req, res) => {

  // request and response handling

});
```

- `req` (request object) contains information sent by the client such as URL, method, and headers.
- `res` (response object) is used to send data back to the client.

The callback function is executed every time a client sends a request to the server.

Step 3: Handle Client Request

Inside the `createServer()` callback, the server processes the client request.

```
console.log(req.url);

console.log(req.method);
```

This helps the server identify which resource is requested and which HTTP method is used.

Step 4: Set HTTP Response Headers

Before sending a response, proper HTTP headers must be set using `writeHead()`.

```
res.writeHead(200, { "Content-Type": "text/plain" });
```

- `200` is the HTTP status code indicating success.
- `Content-Type` specifies the type of data being sent to the client.

Step 5: Send Response to Client

The response body is sent using the `res.write()` or `res.end()` method.

```
res.write("Welcome to Node.js Web Server");

res.end();
```

`res.end()` signals that the response is complete.

Step 6: Make the Server Listen on a Port

Finally, the server must listen on a specific port number to accept client requests.

```
server.listen(3000, () => {
  console.log("Server running on port 3000");
});
```

Handling Multiple URLs (Routing Example)

This example demonstrates basic routing using the http module.

```
const http = require("http");
const server = http.createServer((req, res) => {
  if (req.url === "/") {
    res.writeHead(200, { "Content-Type": "text/plain" });
    res.end("Home Page");
  } else if (req.url === "/about") {
    res.writeHead(200, { "Content-Type": "text/plain" });
    res.end("About Page");
  } else {
    res.writeHead(404, { "Content-Type": "text/plain" });
    res.end("Page Not Found");
  }
});
server.listen(3000);
```

Advantages of Using http Module in Node.js

- No external dependencies
- Lightweight and fast
- Non-blocking request handling
- Suitable for REST APIs
- High performance for concurrent users

Limitations of http Module

- Manual routing required
- More code compared to frameworks
- Not suitable for large applications without frameworks like Express

Real-Time Applications

- Simple web servers
- REST API services
- Microservices
- Backend for single-page applications
- IoT and real-time systems

Creating an HTTP web server in Node.js using the **http module** involves importing the module, creating the server, handling requests, sending responses, and listening on a port. The http module provides a low-level but powerful way to understand how web servers work internally. Although frameworks simplify development, learning server creation using the http module is essential for understanding the fundamentals of Node.js web application development.

5Q. **Discuss Query String and URL processing in Node.js with examples.**

Answer:

In web applications, data is frequently sent from the **client (browser)** to the **server** through a **URL**. This data may represent user inputs such as search keywords, login details, page numbers, filters, etc. In **Node.js**, processing URLs and extracting query string parameters is a common and essential task while developing **web servers**, **REST APIs**, and **dynamic web applications**.

Node.js provides built-in modules such as **`url`** and **`querystring`**, and also exposes URL-related information through the **HTTP request object (`req`)**. Using these features, developers can easily **parse URLs**, **read query parameters**, and **respond dynamically** based on client requests.

A **URL (Uniform Resource Locator)** is the complete address of a resource available on the internet. It specifies **where** the resource is located and **how** it can be accessed.

General Syntax of a URL
protocol://hostname:port/path?querystring#fragment

Example URL
http://localhost:3000/student?roll=101&branch=CSE

Explanation of URL Parts

- **Protocol**: `http`
- **Hostname**: `localhost`
- **Port**: `3000`
- **Path**: `/student`
- **Query String**: `roll=101&branch=CSE`

Query String

A **query string** is the part of the URL that contains **key–value pairs**, used to send data from client to server.

Syntax of Query String

?key1=value1&key2=value2&key3=value3

Example:

?name=Ravi&age=20&course=NodeJS

? → indicates start of query string
& → separates multiple parameters
= → assigns value to a key

Importance of Query String and URL Processing

URL and query string processing in Node.js is used to:

- Read user inputs from browser
- Implement search and filter operations
- Handle pagination
- Create RESTful APIs
- Route requests dynamically
- Pass data between pages without database access

URL Processing in Node.js Using `url` Module

Node.js provides a built-in **`url` module** to parse and process URLs.

Importing `url` Module

```
const url = require("url");
```

url.parse() Method

The `url.parse()` method breaks a URL into readable components.

Syntax: url.parse(urlString, parseQueryString);

- `urlString` → complete URL
- `parseQueryString` → `true` converts query string into object

Example 1: Basic URL Parsing

```
const http = require("http");
const url = require("url");
http.createServer((req, res) => {
  const parsedUrl = url.parse(req.url, true);
  res.writeHead(200, {"Content-Type": "text/plain"});
  res.write("Pathname: " + parsedUrl.pathname + "\n");
  res.write("Query Data: " + JSON.stringify(parsedUrl.query));
  res.end();
}).listen(3000);
```

Request from Browser : (type in the browser address(URL) point):

http://localhost:3000/login?user=admin&pwd=1234

Output:

Pathname: /login
Query Data: {"user":"admin","pwd":"1234"}

- `req.url` contains the full URL part after domain
- `pathname` gives the route
- `query` returns an object containing query parameters.

Query String Processing Using querystring Module

he **`querystring` module** is used to **parse** and **stringify** query strings.

Query String Module:

const querystring = require("querystring");

i) querystring.parse() : Converts query string into JavaScript object.

Syntax:

```
querystring.parse(string);
```

Example:

```
const qs = require("querystring");

const query = "name=Anil&age=21&branch=ECE";
const result = qs.parse(query);

console.log(result);
```

Output:

{ name: 'Anil', age: '21', branch: 'ECE' }

ii) querystring.stringify(): Converts JavaScript object into query string.

Syntax:

```
querystring.stringify(object);
```

Example:

```
const qs = require("querystring");
const obj = { city: "Hyderabad", pin: 500001 };
const result = qs.stringify(obj);
console.log(result);
```

Output:

```
city=Hyderabad&pin=500001
```

URL Processing Using HTTP Request Object (`req`)

In Node.js HTTP server, query parameters can be accessed from req.url.

Example : Extract Query Parameters

```
const http = require("http");
const url = require("url");

http.createServer((req, res) => {
  const data = url.parse(req.url, true).query;

  res.writeHead(200, {"Content-Type": "text/html"});
  res.write("Name: " + data.name + "<br>");
  res.write("Course: " + data.course);
  res.end();
}).listen(3000);
```

Request from Browser : (type in the browser address(URL) point):
http://localhost:3000/?name=Kiran&course=NodeJS

Output:

Name: Kiran

Course: NodeJS

Advantages of Query String and URL Processing

- Simple method to pass data
- No database access required
- Bookmarkable URLs
- Easy debugging
- Suitable for GET requests
- Widely supported by browsers

Query string and URL processing form a **core part of Node.js web development**. By using built-in modules like `url` and `querystring`, Node.js allows efficient extraction and handling of client-side data. Understanding these concepts helps engineering students design **scalable**, **dynamic**, and **interactive web applications**. Proper use of URL parsing improves routing, data handling, and server performance in real-world applications.

6Q). Explain about Node.js core modules: (a) OS Module (b) DNS Module (c) Crypto Module. (d) Util Module definition, syntax and with examples.

Answer:

Node.js Core Modules :

Node.js Core Modules are **built-in modules** provided by Node.js. These modules come **pre-installed**, so there is **no need to install them using npm**. Core modules provide basic functionalities such as operating system information, networking, cryptography, utilities, file handling, etc.

Some important core modules are:

- **os**
- **dns**
- **crypto**
- **util**

(a) OS Module

Definition

The **OS (Operating System) module** provides information about the **operating system** on which the Node.js application is running. It helps developers to know system-level details like memory, CPU, platform, hostname, etc.

Purpose

- Get system information
- Monitor memory usage
- Identify OS type and architecture

Syntax:

```
const os = require("os");
```

Example: OS Module Program:

```
const os = require("os");
console.log("OS Platform:", os.platform());
console.log("OS Architecture:", os.arch());
console.log("Total Memory:", os.totalmem());
console.log("Free Memory:", os.freemem());
console.log("Hostname:", os.hostname());
```

Output::

OS Platform: win32
OS Architecture: x64
Total Memory: 8589934592
Free Memory: 3124576256
Hostname: DESKTOP-PC

Advantages

- Useful for system monitoring
- Helps in platform-dependent applications

(b) DNS Module:

Definition

The **DNS (Domain Name System) module** is used to **resolve domain names into IP addresses** and vice-versa. It enables Node.js applications to perform **network-related name resolution**.

Purpose

- Convert domain names to IP addresses
- Perform network lookups

Syntax:

```
const dns = require("dns");
```

Example: DNS Lookup:

```
const dns = require("dns");
dns.lookup("google.com", (err, address) => {
  if (err) throw err;
  console.log("IP Address:", address);
});
```

Output:

IP Address: 142.250.182.14

Advantages

- Useful in network applications
- Helps in server-side networking

(c) Crypto Module

Definition

The **Crypto module** provides **cryptographic functionalities** such as hashing, encryption, decryption, and secure data handling. It is widely used for **password security and authentication**.

Purpose

- Secure passwords
- Data encryption and hashing
- Generate random values

Syntax:

```
const crypto = require("crypto");
```

Example: Password Hashing using Crypto :

```
const crypto = require("crypto");
const password = "nodejs123";
const hash = crypto
  .createHash("sha256")
  .update(password)
  .digest("hex");
console.log("Encrypted Password:", hash);
```

Output:

Encrypted Password: a1f6c9b9d7...

Advantages

- Improves application security
- One-way encryption (hashing)
- Widely used in login systems

(d) Util Module

Definition

The **Util module** provides **utility functions** that help in debugging, formatting strings, and converting callback-based functions into promise-based functions.

Purpose

- Debugging support
- Object inspection
- Promisify callbacks

Syntax:

```
const util = require("util");
```

util.format():

util.format() is a method of the **Util module** in Node.js used to **format a string by replacing placeholders with given values**, similar to printf() in C. It returns a **formatted string**, which is commonly used for **logging and message construction**.

Example :

```
const util = require("util");
const msg = util.format("Hello %s, your age is %d", "Ravi", 20);
console.log(msg);
```

util.insepect():

util.inspect() is a method of the **Util module** in Node.js used to **convert JavaScript objects into readable string format**. It is mainly used for **debugging** and **logging complex objects**.
It shows:

- Object structure
- Nested properties
- Data types
- Hidden values (if enabled)

Example:

```
const util = require("util");

const student = {  roll: 102,  name: "Anjali",  subjects: {DBMS: 88,    NodeJS: 92}};
console.log(  util.inspect(student, {showHidden: false, depth: null, colors: true}));
```